An adaptable recursive binary entropy coding technique

Aaron B. Kiely and Matthew A. Klimesh

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA

ABSTRACT

We present a novel data compression technique, called recursive interleaved entropy coding, that is based on recursive interleaving of variable-to-variable length binary source codes. A compression module implementing this technique has the same functionality as arithmetic coding and can be used as the "engine" in various data compression algorithms. The encoder compresses a bit sequence by recursively encoding groups of bits that have similar estimated statistics, ordering the output in a way that is suited to the decoder. As a result, the decoder has low complexity. The encoding process for our technique is adaptable in that each bit to be encoded has an associated probability-of-zero estimate that may depend on previously encoded bits; this adaptability allows more effective compression. Recursive interleaved entropy coding may have advantages over arithmetic coding, including most notably the admission of a simple and fast decoder. Much variation is possible in the choice of component codes and in the interleaving structure, yielding coder designs of varying complexity and compression efficiency; coder designs that achieve arbitrarily small redundancy can be produced. We discuss coder design and performance estimation methods. We present practical encoding and decoding algorithms, as well as measured performance results.

Keywords: data compression, binary adaptive coding, recursive interleaved entropy coding

1. INTRODUCTION

In data compression algorithms the need frequently arises to compress a binary sequence in which each bit has some estimated distribution, i.e., probability of being equal to zero. If long runs of bits have nearly identical distributions, then simple source codes, most notably Golomb's runlength codes,[4,5] are quite efficient. However, in many practical situations, not only does the distribution vary from bit to bit, but it is desirable to have the estimated distribution for a bit depend on the values of earlier bits.

Allowing the estimated distribution to change with each new bit can result in more effective compression because a source model can make better use of the immediate context in which a bit appears, and can quickly adapt to changing statistics. For example, when compressing a bit-plane of a wavelet-transformed image one would want to use an entropy coder that can efficiently encode a bit sequence with a probability estimate that varies from bit to bit.

Accommodating a dynamically changing probability estimate is tricky because the decompressor will need to make the same estimates as the compressor. In general, before the ith bit can be decoded, the values of the first $i-1$ bits must be determined. This requirement makes it difficult to efficiently use simple source codes such as runlength codes. To our knowledge, currently the only efficient coding methods that accommodate a bit-wise adaptive probability estimate are arithmetic coding*[7,9,11] and a relatively obscure technique called interleaved entropy coding.[14–16]

In this paper, we describe a new technique called *recursive interleaved entropy coding*, which is a generalization of interleaved entropy coding. A recursive interleaved entropy coder compresses a binary source with a bit-wise adaptive probability estimate by recursively encoding groups of bits with similar distributions, ordering the output in a way that is suited to the decoder. Much variation is possible in the choice of component codes and in the interleaving structure, yielding coder designs of varying complexity and compression efficiency.

Further author information: E-mail: {aaron, klimesh}@shannon.jpl.nasa.gov, Address: Jet Propulsion Laboratory, 4800 Oak Grove Drive, Mail Stop 238-420, Pasadena, CA 91109, USA

*We include in the family of arithmetic coders assorted approximate arithmetic coders such as the quasi-arithmetic coder of Ref. 3, and the Z-coder.[13]

The functionality of recursive interleaved entropy coding is essentially the same as that of binary arithmetic coding[6–9]; however, there are many practical differences. Arithmetic encoding of one bit requires a few arithmetic operations and usually at least one multiplication. Our encoder requires no arithmetic operations except those which might be needed to choose a code index based on the estimated bit distribution; however it requires some bookkeeping and bit manipulation operations. Our encoder requires more memory than arithmetic coding. Arithmetic decoders are generally of similar complexity to the encoders, but our decoder is much simpler than our encoder: it needs fewer operations than the encoder, and requires only a small amount of memory.

A more complete version of the results presented here is contained in Ref. 1. In a related paper,[2] we describe modified encoding and decoding techniques with lower encoder memory requirements.

In the remainder of this section, we give a brief overview of the entropy coding technique. Section 2 describes the details of encoder and decoder operation and presents practical encoding and decoding algorithms. In Sect. 3 we examine a class of binary trees that appear to be well suited for use in coder designs. Section 4 gives methods for estimating the performance of a given coder design. In Sect. 5 we describe a technique for designing an encoder that meets a given redundancy constraint. Section 6 provides performance results. Finally, Sect. 7 provides a conclusion and identifies some open problems.

1.1. The Source Coding Problem

We examine the problem of compressing a sequence of bits $b_1, b_2, \ldots$ from a random source. For each source bit b_i we have a probability estimate $p_i = \text{Prob}[b_i = 0]$ which may depend on the values of the source sequence prior to index i, and on any other information available to both the compressor and decompressor. This dependence encompasses both adaptive probability estimation as well as correlations or memory in the source. Because of this dependence, efficient coding requires a bit-wise adaptable coder. We are not concerned here with methods of modeling the source, and so we make no distinction between the actual and estimated probabilities.

Without loss of generality, we assume that $p_i \geq 1/2$ for each index i. If this were not the case for some p_i, we could simply invert bit b_i before encoding to make it so; this inversion can clearly be duplicated in the decoder.

We also assume that the decompressor can determine when decoding is complete. In practice, this often occurs automatically; otherwise, a straightforward method such as transmitting the sequence length prior to the compressed sequence can be used.

Although we only discuss the compression of binary sequences, given any nonbinary source we can assign prefix-free binary codewords to source symbols to produce a binary stream. Thus a bit-wise adaptable coder such as the one we describe here can be applied to nonbinary sources as well.

1.2. The Recursive Interleaved Entropy Coder Concept

We now give an overview of how recursive interleaved entropy coding works. To simplify the explanation, we defer some of the details until Sect. 2.

Since, by assumption, each bit has probability-of-zero at least 1/2, we are concerned with the probability region $[1/2, 1]$. We partition this region into several narrow intervals, and with each interval we associate a bin that will be used to store a list of bits. When bit b_i arrives, we place it into the bin corresponding to the interval containing p_i. Because each interval spans a small probability range, all of the bits in a given bin have nearly the same probability-of-zero, and we can think of each bin as corresponding to some nominal probability value.

Bits in the leftmost bin, whose interval contains probability 1/2, do not undergo further processing. For every other bin we specify an exhaustive prefix-free set of binary codewords. When the bits collected in a bin form one of these codewords, we delete these bits from the bin and encode the value of the codeword by placing one or more new bits in other bins[†]. In effect, bits in a bin are encoded using a prefix-free variable-to-variable length code, with the added twist that the output bits are assigned to other bins where they may be further encoded.

[†]The rules for collecting bits to form a codeword are not straightforward and we save the details for Sect. 2.2.

The mapping from codewords to encoded bits is conveniently described using a binary tree. Each codeword is assigned to a terminal node in the tree, non-terminal nodes are labeled with a probability value that determines a destination bin, and the branch labels (each a zero or one) indicate the output bits that are placed in the destination bins. For example, Fig. 1 shows a tree that might be used for a bin with nominal probability 0.9. The prefix-free codeword set for this bin is $\{00, 01, 1\}$, shown as labels of the terminal nodes in the tree. If the codeword to be processed in the bin is 00, which occurs with probability approximately 0.81, we place a zero in the bin that contains probability 0.81. If the codeword is 1, first we place a one in the bin containing probability 0.81, which indicates that the codeword is something other than 00, then we place a zero in the bin containing probability 0.53 because, given that the codeword is not 00, the conditional probability that the codeword is 1 is approximately 0.53. We can see that this process contributes to data compression because the most likely codeword is 00, which is represented using a single bit.

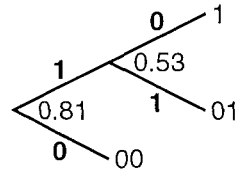

Figure 1: Example of a tree for a bin with representative probability 0.9.

Bits that reach the leftmost bin form the encoder's output; we refer to this bin as the "uncoded" bin since these bits do not undergo further coding. These bits are zero with probability very close to 1/2 and are thus nearly incompressible, so leaving these bits uncoded does not add much redundancy.

During the encoding process, bits arrive in various bins either directly from the source or as a result of processing[‡] codewords in other bins. Our goal is to have bits flow to the leftmost bin. To accomplish this, we impose the constraint on the non-terminal node labels that all new bits resulting from the processing of a codeword must be placed in bins strictly to the left of the bin in which the codeword was formed. Apart from our desire to move bits to the left, this constraint also prevents encoded information from traveling in "loops", which would make coding difficult or impossible.

As illustrated in the example above, a natural method of mapping output bits to bins is to assign each output bit of a codeword to the bin indicated by the output bit's probability-of-zero, as computed from the nominal probability of the bin in which the codeword is formed. If we use this method, then a bin with nominal probability p must necessarily use a tree that is "useful" at p according to the following definition:

Definition:

1. *We say that a tree is* useful at probability p *if it has the property that when all input bits have probability-of-zero equal to p, all output bits have probability-of-zero in the range* $[1/2, p)$.
2. *A tree is* useful *if there exists some p for which the tree is useful at p.*
3. *If the branches of a tree lack output bit labels, then we say that the tree is* useful *(resp.* useful at probability p*) if some assignment of output bit labels makes the tree useful (resp. useful at probability p).*

Perhaps surprisingly, requiring output bits to be mapped strictly to the left turns out to be a reasonable constraint — we'll see in Sect. 3.1 that for any $p \in (1/2, 1)$ there exists a tree that is useful at p.

In practice, bins are identified by indices rather than nominal probability values, starting with index 1 for the leftmost bin. At each non-terminal node in a tree we identify the index, rather than the nominal probability value, of the bin to which the associated output bit is mapped. The constraint on encoder design is now that

[‡]We refer to the encoding of a codeword in a bin as "processing" the codeword, rather than encoding, to avoid confusion with the overall encoding procedure.

each output bit from the tree for bin j must be mapped to a bin with index strictly less than j. No computations involving probability values are needed for encoding apart from those which may be required to map input bits to the appropriate bins.

The mapping of output bits to bins can be designed without regard for nominal probability values, so it is possible to design working coders that include trees which are not useful. In fact, good coders can be designed that do not contain useful trees. However, one expects better compression if output bit probabilities are in close agreement with their destination bins' nominal probabilities; this appears to be most easily achieved by exploiting useful trees.

Near the end of encoding a sequence of bits there may be bins that contain partial codewords that must be "flushed" from the encoder. When this occurs, we append one or more extra bits to the partial codeword to form a complete codeword which is then processed in the normal manner.

Clearly the encoder's output contains some redundancy because source bits with slightly different probabilities-of-zero are treated the same; that is, the bins' intervals have positive widths. As one might expect, by increasing the number of bins in the coder design we can decrease the redundancy to arbitrarily small values. This result is formalized in Sect. 3.2.

1.3. Relation to Interleaved Entropy Codes

An important special case of our entropy coder arises when all output bits generated from each tree are mapped to the uncoded bin. In this case the coder essentially interleaves several separate variable-to-variable length binary codes. This technique was first suggested in Ref. 15, where Golomb codes are interleaved, and has also appeared in Ref. 16. Howard[14] gives a more thorough analysis of this technique, which we refer to as non-recursive interleaved entropy coding. A non-recursive coder design allows reduced complexity encoding; in Sect. 6 we show that in fact non-recursive coders can achieve very high encoding speeds.

By using an increasing number of increasingly complex variable-to-variable length codes, it's clear that we can make asymptotic redundancy arbitrarily small with a non-recursive coder (provided that the estimates of the source distribution can be made arbitrarily accurate). With the additional flexibility of the recursive technique presented here, a given redundancy target tends to be achievable with fewer and/or simpler codes.

2. ENCODING AND DECODING

Section 1 gave an overview of how recursive interleaved entropy coding works. In this section we describe the encoding and decoding procedures in more detail and give practical algorithms for encoding and decoding.

It should be noted that the encoding algorithm presented here requires memory resources that are proportional to the length of the source bit sequence. In Ref. 2 we describe alternative encoding algorithms (with corresponding decoding algorithms) that have much more modest memory requirements.

We first state more precisely how a coder is specified. A recursive interleaved entropy coder specification consists of:

1. A rule for mapping source bits to bins that are numbered from 1 to B.
2. For each bin with index greater than 1, an exhaustive prefix-free set of binary codewords and a binary tree that describes rules for processing each codeword by placing one or more bits in lower-indexed bins.

While we usually think of each bin in the encoder as corresponding to some probability interval, such a relationship is not required, and there need not be any implicit probability estimate used to map bits to bins.

Since we are not concerned with source modeling, we will frequently present coder designs without specifying a rule for mapping source bits to bins. In this case we assume that each source bit is mapped to the bin that minimizes redundancy given the source bit probability-of-zero p_i.

As a running example to illustrate the encoding and decoding procedures, we use the following five bin coder design which we refer to as C5:

Coder Design C5: This coder design is illustrated in Fig. 2, where the trees shown identify the relationship between codewords and output bits. For example, Fig. 2(c) indicates that if codeword 01 is formed in bin 4 then we place bits 1,0,1 in bins 3,2,1 respectively.

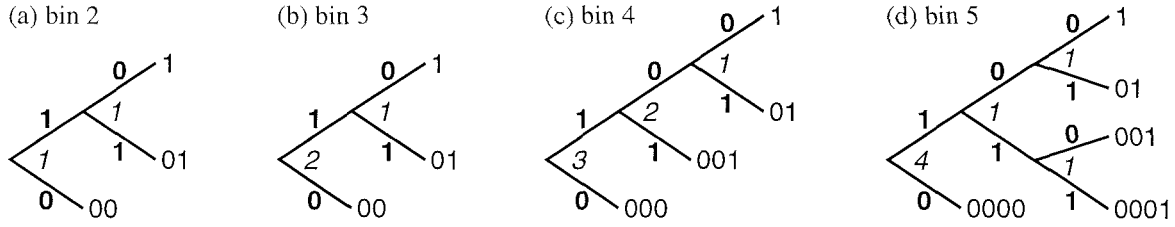

Figure 2. The five bin coder design C5. Output bits are shown in boldface, the corresponding bin indices are in italics. The input codewords are shown at terminal nodes of the trees. The first bin of a coder design does not have an associated tree.

2.1. Decoder Operation

We first describe the decoding procedure since it determines how encoding must be performed. We regard each bin in the decoder as containing a list of bits. To begin, all of the encoded bits are placed in the first (uncoded) bin, and all other bins are empty. At any time, each nonempty bin (with the exception of the uncoded bin) will contain a single codeword or a suffix of a codeword. To decode a source bit, we take the next bit from the bin to which the source bit was assigned. If this bin is empty, we first reconstruct the codeword in that bin by taking bits from other bins as needed. This in turn may require reconstructing codewords in those bins, and so on.

Decoding can be accomplished using two recursive procedures, `GetBit` and `GetCodeword`, shown in Fig. 3. `GetBit` simply takes the next available bit from the indicated bin. If this bin is empty then it first calls `GetCodeword`. Given an empty bin, `GetCodeword` determines which codeword must have occupied the bin by taking bits from other bins (via `GetBit`), and then places that codeword in the bin. The `GetCodeword` procedure is similar to Huffman decoding, except that at each step we take the next bit from the appropriate bin, not (necessarily) from the encoded bit stream.

To decode the ith source bit, let `binindex` equal the index of the bin to which this source bit is assigned according to the bin assignment rule for the coder. Then the ith decoded bit is equal to `GetBit(binindex)`.

2.2. Encoder Operation

The encoding procedure outlined in Sect. 1.2 illustrated the codeword processing operations involved in encoding, but did not discuss the order in which bits are collected to form codewords. This order is important; we must encode in a way that allows the decoder to determine the value of source bit b_{i-1} before attempting to decode source bit b_i, since the bin to which a source bit is mapped may depend on the values of previous source bits.

During encoding, each bin is viewed as containing a list of bits, but when a bit arrives in a bin as an output bit from another bin, the bit might be inserted into the list at a position other than the end of the list.

One method of encoding is to first place all input bits in the appropriate bins, then process codewords starting with those in the highest-indexed bin and working toward bin 1. At each step we identify the nonempty bin with the largest index and we take bits (in priority order) from this bin until we have formed a codeword, appending flush bits if needed to complete the final codeword of the bin.

For software encoding, we maintain a linked list of bit values. Each record in the list stores a bit value and the index of the bin that contains the bit. Initially the list contains the entire input sequence in order of arrival. When a codeword is processed, we delete the bits that formed the codeword and insert the resulting output bits in the list at the location of the first bit in the codeword.

For example, suppose an encoder using design C5 has its linked list in the state shown in the left half of Fig. 4. The largest bin index in this list is 4, so we search through the list for bits in bin 4 until we form the

`GetBit(binindex)`:

1. If the bin with index `binindex` is empty then call `GetCodeword(binindex)`.
2. Remove the first bit in the bin with index `binindex` and return the value of this bit.

`GetCodeword(binindex)`:

1. Initialize `nodepointer` to point to the root of the tree for the bin with index `binindex`.
2. While `nodepointer` is not pointing to a terminal node, do the following:
   (a) Let `thisbinindex` equal the bin index indicated by the node at `nodepointer`.
   (b) Assign `bitvalue` = `GetBit(thisbinindex)`.
   (c) Let `nodepointer` point to the node indicated by the branch with label equal to `bitvalue`.
3. At this point `nodepointer` points to a terminal node, which means that we have reached a codeword. Place this codeword in bin `binindex`.

Figure 3: The `GetBit` and `GetCodeword` procedures used for recursive decoding

codeword 01. This codeword produces output bits 1,0,1 in bins 3,2,1 respectively; the new state of the linked list is shown in the right half of Fig. 4.

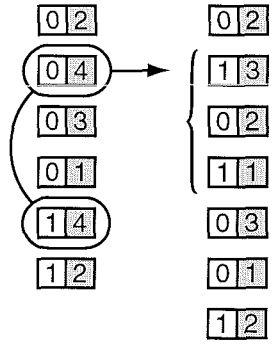

Figure 4. One step of software encoding using coder design C5. In each pair, the left (unshaded) box indicates bit value; the right (shaded) box shows bin index.

When all bits are in the first bin, the encoder output consists of these bits in priority order.

Ref. 1 gives a set of rules to which an encoder must conform, and describes other encoding techniques that conform to these rules. Encoding variations amount to changing the order in which certain processing operations are performed, and using different methods for identifying codewords to be processed. The encoding methods all yield the same encoded bit stream, apart from differences arising due to different choices of flush bits. In Ref. 2 we describe encoding methods that make more efficient use of memory.

We now turn to the task of assigning flush bits when needed to complete the last codeword in a bin. When they are required, any choice of flush bits that forms a complete codeword will produce a decodable sequence. The decoder does not need to know the encoder's method of selecting flush bits — they are simply the bits remaining in the decoder after decoding is complete. This means that decoder speed is essentially unaffected by whether the encoder uses a quick or a highly optimized method to choose such bits.

We would like to choose flush bits in a way that minimizes the length of the encoded sequence. however, in general the optimal assignment of flush bits in a given bin may depend on the contents of lower-indexed bins. In Ref. 1 we describe a simple but suboptimal greedy rule for assigning flush bits. The greedy rule allows one to assign flush bits during encoding using a look-up table and seems to produce an encoded length that is at worst only a few bits longer than the length obtained from optimal flush bit assignment.

3. USEFUL TREES

Recall from Sect. 1 that a tree whose branches lack output bit labels is useful at probability p if some assignment of output bit labels results in all output bits having probability-of-zero in the range $[1/2, p)$ when the input bits all have probability-of-zero equal to p.

From the overview of coder operation given in Sect. 1, we expect a bin to be associated with some probability interval, and we require that all output bits generated at a bin must be mapped to bins with strictly lower indices, which we expect to be associated with smaller probability values. Thus, if there are probability values for which useful trees do not exist, it may be difficult to produce coder designs achieving very small redundancies.

Fortunately, it turns out that there is essentially no restriction on the probability values at which a useful tree can be found. There exist families of trees such that for any $p \in (1/2, 1)$, there is a tree in the family that is useful at p. We call such a family *complete*. A complete family of trees is necessarily infinite.

3.1. Useful Trees Exist Everywhere

We now describe a particular family of useful trees. For integers $n > m \geq 1$ we form the tree $\mathcal{T}_{m,n}$ as shown in Fig. 5[§].

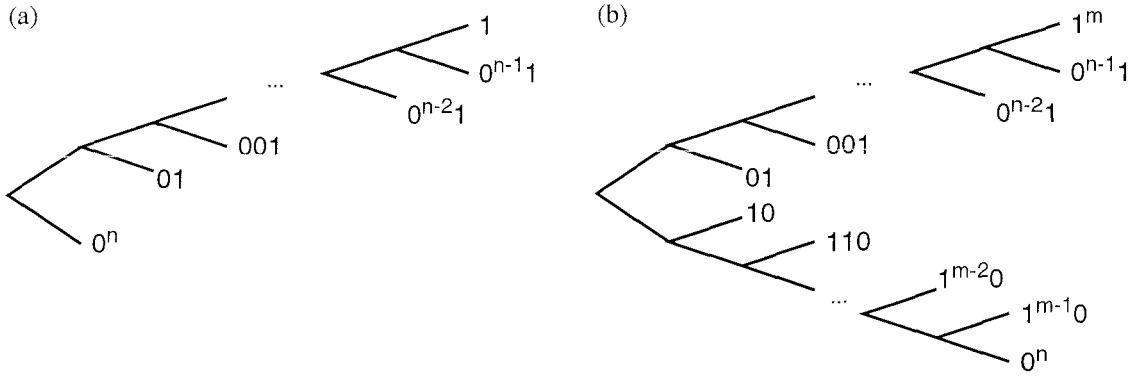

Figure 5: The tree $\mathcal{T}_{m,n}$ (a) for $m = 1$, and (b) for $m > 1$.

For $m = 1$ and $n > m$, $\mathcal{T}_{m,n}$ is useful in the interval $(\gamma_{m/n}, 1)$. For $n > m > 1$, $\mathcal{T}_{m,n}$ is useful in the interval $(\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$. Here, for $\alpha \in [0, 1)$, γ_α denotes the root of $p = (1-p)^\alpha$ that is in the range $(1/2, 1]$.

It can be verified that $\gamma_{m/n}$ decreases as m/n approaches 1, and that $\gamma_{m/n}$ approaches $1/2$ as m/n approaches 1. Consequently, for any fixed integer $d > 0$, the set $\{\mathcal{T}_{m,m+d}\}_{m=1}^{\infty}$ forms a complete family of useful trees. Figure 6 illustrates how the useful regions for the trees in the important family $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$ span $(1/2, 1)$.

Some other examples of useful trees are given in Ref. 1.

[§]The authors gratefully acknowledge Sam Dolinar for pointing out this generalization of the $\mathcal{T}_{m,m+1}$ trees.

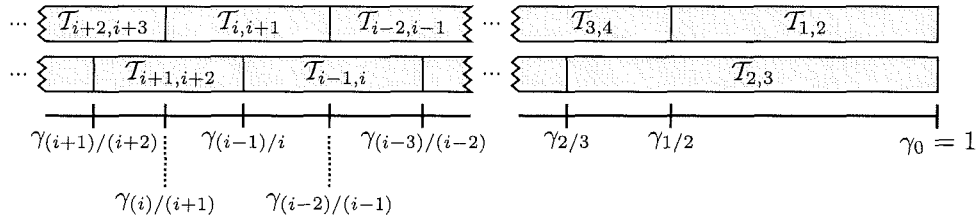

Figure 6: Useful regions of trees in the family $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$. (Not to scale.)

3.2. Application of Useful Trees to Coder Design

We noted in Sect. 1.3 that redundancy can be made arbitrarily small (subject to the accuracy of the source distribution estimates) using nonrecursive interleaved entropy coders; however the codes required quickly become complex. On the other hand, we saw in Sect. 3.1 that we can easily construct a complete family of useful trees with manageable complexity. The following theorem, which we prove in Ref. 1, establishes that such a family of trees provides another method of producing coder designs achieving arbitrarily small redundancy when we exploit recursion in the coder.

THEOREM 3.1. *Let $\mathcal{U}$ be a complete family of useful trees without output bit specifications. Then for any $\varepsilon > 0$ and $\delta > 0$ there exists a coder design that uses only trees from $\mathcal{U}$, and a constant c, for which the following holds: For any n and any sequence of bits $b_1, \ldots, b_n$ whose associated probability-of-zero estimates $p_1, \ldots, p_n$ are all in the range $[\delta, 1-\delta]$, the coder will compress the sequence to at most*

$$c + (1+\varepsilon) \sum_{i=1}^{n} \left\{ \begin{array}{ll} -\log_2 p_i, & \text{if } b_i = 0 \\ -\log_2(1-p_i), & \text{if } b_i = 1 \end{array} \right\} \text{ bits.} \tag{1}$$

It should be emphasized that this theorem is not probabilistic and there need not be any relation between the bit values and their probability-of-zero estimates. However, if, for example, we generate independent random bits $b_1, \ldots, b_n$ and let $p_i = \text{Prob}[b_i = 0]$, then the theorem implies that the expected number of encoded bits per source bit can be made to approach the source entropy. Similar results hold under more complicated assumptions, so long as the probability estimates are accurate for each bit. Even if the estimates are not accurate, the sum in Eq. (1) represents, in a rather loose sense, the best average encoded length achievable by a coder which relies on the estimates.[6] An ideal arithmetic coder that uses the same probability estimates would produce encoded length slightly larger than the sum in Eq. (1).

Using Thm. 3.1 it can be shown that redundancy can be made arbitrarily small in other senses as well. For example, assuming accurate probability estimates, the average number of bits of redundancy per source bit can be made arbitrarily small without any restriction on the source bit probabilities.

4. ESTIMATING COMPRESSION EFFICIENCY

In this section we turn to the problem of quantifying the compression efficiency of a recursive interleaved entropy coder. We would like to analytically determine the redundancy resulting from a given coder design, but this depends on the source and does not appear to be easy to calculate exactly in general.

One metric that gives a good indication of performance, and for which we can find reasonable estimates, is the rate (the expected number of output bits per source bit) when the input to the encoder is an IID stream of bits into bin j, each bit having probability-of-zero equal to p. We denote this quantity by $R_j(p)$.

Because of the recursive nature of the encoder, our estimates of $R_j(p)$ rely on rate estimates for other bins. A given bin may have bits with different probabilities-of-zero arriving from higher-indexed bins. If bin j has as input λ_1 bits each with probability-of-zero q_1 and λ_2 bits each with probability-of-zero q_2, then the resulting contribution to the number of output bits might be approximated as

$$\lambda_1 R_j(q_1) + \lambda_2 R_j(q_2) \tag{2}$$

or

$$(\lambda_1 + \lambda_2) R_j \left(\frac{\lambda_1 q_1 + \lambda_2 q_2}{\lambda_1 + \lambda_2} \right). \tag{3}$$

The first approximation would tend to be more accurate when long runs of bits in bin j have the same probability-of-zero, and the second would be more accurate if the two types of bits are well mixed.

In this section we describe two techniques for estimating $R_j(p)$ that are direct applications of the respective approximations above. Extensions of these techniques can be used to accurately estimate the rate obtained for a source that produces bits with varying (but known) distributions. The rate estimates produced are asymptotic as the input sequence length becomes large, i.e., the cost of bits used to flush the encoder is not included.

The rate estimation techniques do not usually give exact results, in part because bits arriving in a bin may not be independent even when the source bits are independent. This dependence can arise, e.g., when processing a single codeword results in more than one output bit being placed in the same bin. In practice, however, both techniques usually give quite good estimates.

4.1. Two Rate Estimation Methods

In our first method of rate estimation, we estimate $R_j(p)$ from the estimates for $R_1(p)$, $R_2(p), \ldots, R_{j-1}(p)$. For non-terminal node k of the tree for bin j, let $\eta_k(p)$ denote the expected number of output bits per input bit, $q_k(p)$ denote the probability-of-zero of these bits, and B_k denote the destination bin of these bits. We use the estimate

$$R_j(p) = \begin{cases} \sum_k \eta_k(p) R_{B_k}(q_k(p)), & \text{if } j > 1 \\ 1, & \text{if } j = 1, \end{cases} \tag{4}$$

where the sum is over all non-terminal nodes in the tree.

Our second technique for estimating $R_j(p)$ is based on Eq. (3). For each bin we produce a list of (λ, q) pairs; where in each pair λ represents and expected number of bits per source bit and q is the corresponding probability-of-zero. Initially each list is empty except the list for bin j, which contains the pair $(1, p)$.

We now let ℓ step through bin indices, starting with $\ell = j$ and working down through $\ell = 2$. At a given step, suppose the list for bin ℓ contains pairs (λ_1, q_1), $(\lambda_2, q_2), \ldots, (\lambda_m, q_m)$. We compute the total expected number of bits in the bin per source bit,

$$\Lambda_\ell = \sum_{i=1}^{m} \lambda_i, \tag{5}$$

and the average probability-of-zero in the bin,

$$Q_\ell = \frac{1}{\Lambda_\ell} \sum_{i=1}^{m} q_i \lambda_i.$$

Treating the input to bin ℓ as Λ_ℓ bits, each with probability-of-zero Q_ℓ, we compute the expected number of bits λ'_k and associated probability-of-zero q'_k at each non-terminal node k in the tree and append (λ'_k, q'_k) to the list for the bin to which the output bit associated with node k is mapped[¶]. Then we decrement ℓ to move to the next bin.

After we finish stepping through the bins, our estimate of $R_j(p)$ equals Λ_1, the total expected number of bits in the first bin computed using Eq. (5).

This technique can easily be adapted to estimate the rate associated with a source that produces bits with varying (but known) distributions. For a source that produces bits with probabilities-of-zero $\phi_1, \ldots, \phi_k$ with frequencies $f_1, \ldots, f_k$ we simply initialize our lists so that each pair (f_i, ϕ_i) is put in the list for the bin to which probability ϕ_i is mapped.

[¶]If bits generated at node k are assigned to bin 1, then computation of q'_k is not necessary.

4.2. Performance Example

The rate estimation techniques of Sect. 4.1 allow computation of estimates of the redundancy $\rho_j(p) = R_j(p) - \mathcal{H}_2(p)$ for each bin j of a coder. We typically plot redundancy for a coder design as a function of p, under the assumption that source bits are mapped to the bin that minimizes the estimated rate. Generally, each $R_j(p)$ is nearly linear in p, so $\min_j R_j(p)$ is nearly piecewise-linear. The rate function $\min_j R_j(p)$ tends to hug the binary entropy curve, resulting in a redundancy curve with a "sawtooth" appearance.

To demonstrate the accuracy of the estimation techniques, we have estimated redundancy for coder design C5 and measured the actual redundancy by simulation. Figure 7 shows the results. The results of the two estimation techniques are indistinguishable at the scale of the figure, and are in close agreement with the measured redundancy. For bins 1–3, both redundancy estimates can be shown to be equal to the actual redundancy.

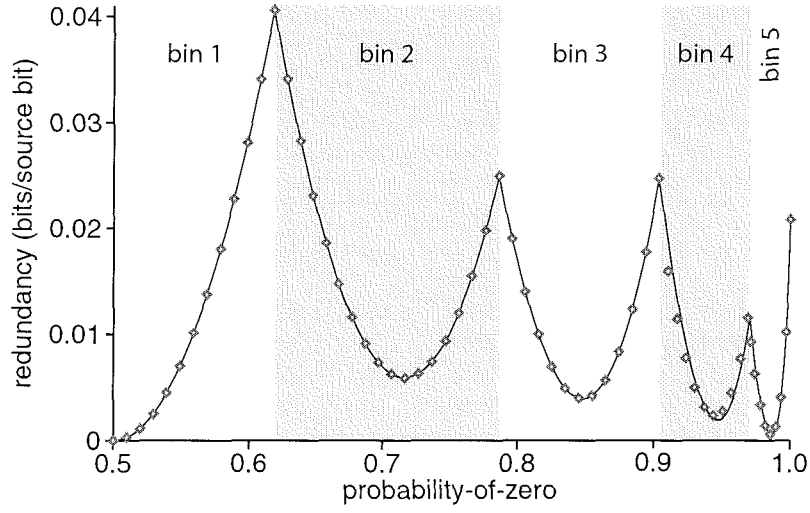

Figure 7. Estimated (solid curve) and measured (individual points) redundancy for coder design C5. Each point was generated using 500 sequences, each of length 2^{20} bits.

5. DESIGNING CODERS

In this section we describe a practical procedure for finding a coder design whose redundancy meets a given goal and we exhibit some coder designs obtained using this design procedure. As discussed in Sect. 4, evaluating the exact redundancy of a given coder design is a difficult problem. Thus, we consider only the redundancy obtained for sources with a fixed probability-of-zero, and we rely on the rate estimates of Sect. 4, which in practice are quite close to the rates achieved.

The redundancy goal is specified by a quantity Δ that represents the maximum allowed asymptotic redundancy, in bits per source bit. The procedure requires a set of candidate trees to be used in the coder. In this context the trees do not include assignments of bin indices to non-terminal nodes or output bit labels to branches; these assignments will be made as part of the design procedure. To use the design procedure with arbitrarily small values of Δ, the set of candidate trees should have the property that for any $p \in (1/2, 1)$ some tree in the set is useful at p — that is, the set contains a complete family of useful trees. In practice, it is often convenient to limit consideration to a finite candidate tree set.

Each bin j will have associated with it an interval specified by the left endpoint z_{j-1} and right endpoint z_j. For $j > 1$ a tree will also be associated with bin j. No design work is required for bin 1, since $z_0 = 1/2$ and bin 1 is uncoded. To design to rest of the coder, we add bin specifications in order of increasing bin index by selecting a tree, assigning bin indices and output bit labels to the tree, and computing the left endpoint of the bin's interval. For example, Fig. 8 shows a case where the coder design has been specified for the first three bins,

and our redundancy target Δ is met for source probabilities-of-zero less than p^*. The next task is to specify a tree that meets the redundancy target for an interval that includes p^*; as a result we will have $z_3 \leq p^*$.

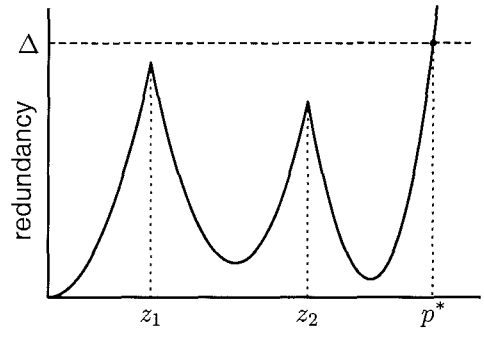

Figure 8: Redundancy of a coder after specifying trees for bins 2 and 3.

Given that the first $j-1$ bins of the coder satisfy the redundancy target up to probability-of-zero p^*, the tree for bin j needs to yield redundancy of at most Δ at p^*. This can be accomplished by selecting a tree that is useful at p^* and, treating the input bits to the tree as all having probability-of-zero p^*, assigning branch labels so that each output bit is more likely to be a 0 than a 1. Then, at each non-terminal node we assign the output bin index ℓ so that the probability-of-zero of the output bit is in the interval $[z_{\ell-1}, z_\ell)$ (for this assignment we temporarily let $z_j = p^*$). This construction maps output bits to bins in regions where the redundancy is less than the target Δ, and it can be shown that, (to the extent that Eq. (4) is accurate), the tree we have selected for the new bin produces redundancy less than Δ at p^*. Since the rate function for each bin is continuous, we have extended the range where the coder meets the redundancy target.

The above procedure can always be used to find one or more trees (labeled with output bits and destination bins) that extend the range over which the coder meets the redundancy target. However, additional trees, or trees with alternate output bit and bin assignments, might also meet the redundancy goal at p^*. A possible method of finding such trees is to select a tree and assign bins and output bits as in the above procedure, but using a target probability somewhat larger than p^*.

A reasonable method of choosing among several trees which meet the redundancy goal at p^* is to pick the tree which extends the useful range of the coder the furthest. The procedure is then essentially a greedy algorithm, and there is no guarantee that the number of bins in the coder will be minimized. In addition, the design procedure does not give consideration to minimizing encoding or decoding complexity, which may be quite different than minimizing the number of bins.

6. ENCODING AND DECODING SPEED

We now examine the encoding and decoding speeds obtained from coders of various compression efficiencies. We have measured the encoding and decoding speeds of software implementations of our coding technique and of arithmetic coding. We have also measured the redundancy achieved by the coders. All timing tests were performed on a Sun Ultra Enterprise with a 167 MHz UltraSPARC processor.

For our test source sequences, we generated sequences of 500,000 probability values p_i from a uniform distribution on $[0, 1]$, and produced random bits b_i according to these values. We compute bin assignments outside of the timing loop as the optimal assignment given p_i to isolate the speed and efficiency of the actual coding from the source modeling.

For comparison, we also evaluated the "shift/add" binary arithmetic coder from Ref. 11 with parameters $b = 16$, and several different values of f (see Ref. 11 for definitions of these parameters). The arithmetic coder was modified to be similarly isolated from the modeling; bit probabilities were supplied in a form convenient to the coder. We selected the coder from Ref. 11 because it is reasonably fast, it is widely used by other researchers,

the source code is publicly available, and it was relatively easy to isolate the coder from the source modeling. Other arithmetic coder implementations (e.g. Ref. 10) may be somewhat faster.

Figure 9 shows redundancy versus decoding speed for two recursive coder designs as well as the arithmetic coder. The recursive coder designs are the 6-bin and 10-bin coders specified in Tables C-3 and C-2 respectively of Ref. 1. The 6-bin coder design yields a fast decoder in part because the coder design is recursive only in the last bin; note however that our software does not explicitly take advantage of this property. Figure 9 shows that recursive interleaved entropy coding can offer a noticeable improvement in decoding speed over arithmetic coding. For many data compression applications, e.g., image retrieval from databases, decoding speed is much more important than encoding speed, and recursive interleaved entropy coding appears to be an attractive alternative to arithmetic coding in such a situation.

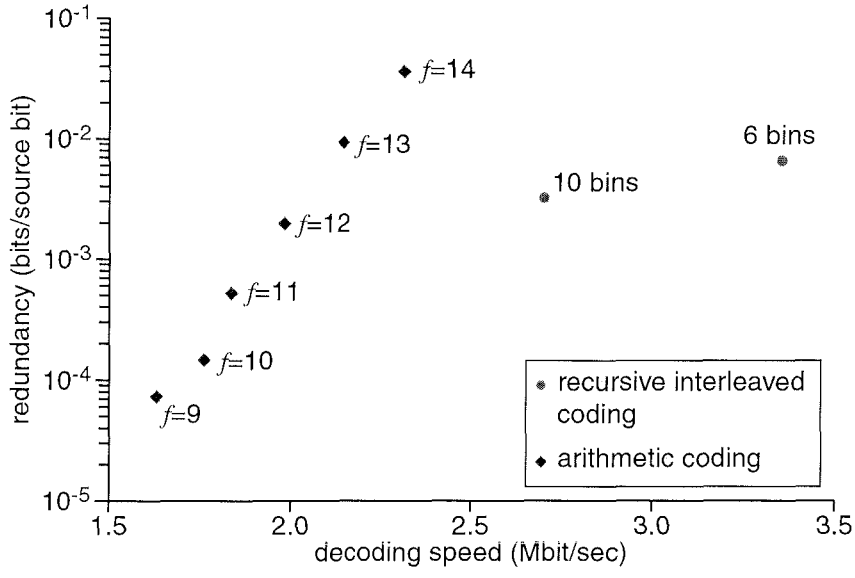

Figure 9: Decoding performance of two recursive coder designs and the arithmetic coder from Ref. 11.

For spacecraft applications, however, encoding speed is paramount. Figure 10 shows redundancy versus encoding speed for the two recursive coder designs tested above, for two non-recursive coder designs with a specialized encoder, and for the arithmetic coder. We observe that the recursive coders offer encoding speed comparable to that of arithmetic coding. For a recursive coder design, we expect that encoding is inherently slower than decoding.

The non-recursive coder designs in Fig. 10 are specified in Table C-3 of Ref. 1. Here encoding speed is measured for an encoder specifically tailored to exploit non-recursive designs. We see from the figure that the non-recursive coder designs shown here provide encoding that is more than twice as fast as that of the arithmetic coder. As noted earlier, non-recursive interleaved entropy coders have been investigated in Refs. 14–16; however, previous implementations have used less general component codes than our implementations and the potential for fast encoding (and decoding) does not appear to have been fully appreciated.

A non-recursive coder design that meets a given redundancy target requires larger trees, and usually a larger number of bins, than a recursive design. For example, achieving low redundancy when the source has probability-of-zero close to one requires the use of very large trees in non-recursive coders, but not in recursive coders. However, at the present stage of development non-recursive coders appear to have an advantage when encoding speed is our primary concern.

These results give some indication of the performance achievable using recursive and non-recursive interleaved entropy coding. We have good reason to be optimistic that even better coder designs are possible. First, note that our coder design procedure essentially ignores encoding and decoding complexity. Second, the number of

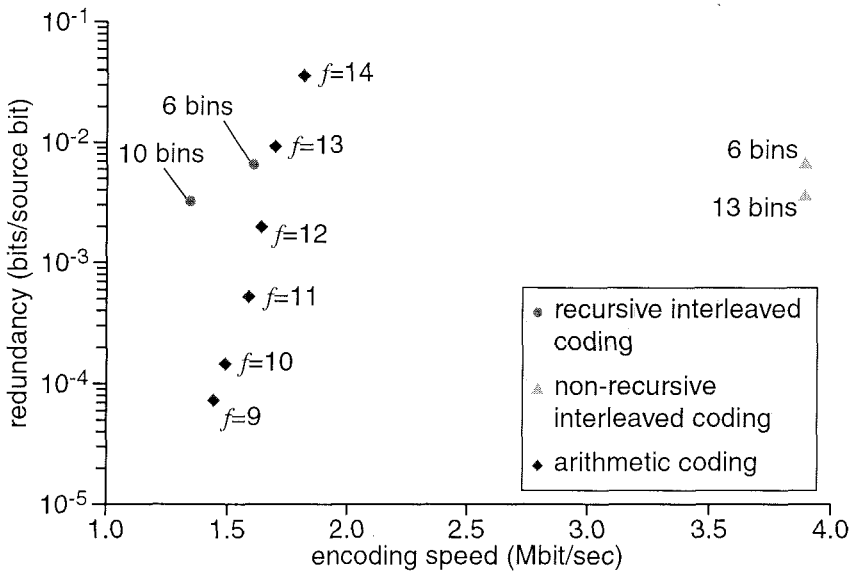

Figure 10. Encoding performance of two recursive coder designs, two non-recursive coder designs, and the arithmetic coder from Ref. 11.

useful trees grows very quickly as we increase the size of the tree, and from Sect. 1.2 that good coder designs may include trees that are not useful. Thus the number of potential coder designs quickly becomes large as we increase the size of the candidate trees.

7. CONCLUSION

We have presented a new entropy coding technique that provides the same functionality as binary arithmetic coding. The technique accommodates an adaptive probability estimate, which allows a data compression algorithm to exploit sophisticated source models, enabling efficient compression. The technique can in theory achieve arbitrarily small asymptotic redundancy as coder designs increase in complexity. We have described a rate estimation technique and a practical coder design procedure. Using the design procedure we have found relatively simple coder designs that yield efficient compression. Compared to arithmetic coding, our technique provides competitive encoding speed and noticeably better decoding speed. For the special case of a non-recursive coder design, we can achieve significantly faster encoding than with arithmetic coding.

We see that recursive interleaved entropy coding appears to be a viable alternative to arithmetic coding. As recursive interleaved entropy coding is still a very new technique, it is reasonable to expect further improvements, perhaps from both discovering better coder designs and developing algorithmic improvements. By comparison, arithmetic coding has been maturing for over two decades. Our encoding speed tests of non-recursive interleaved entropy coding indicate that the extent of the benefits of that technique has not been previously appreciated.

Several interesting directions for future research remain. Although we have exhibited a practical coder design procedure, it is likely that this procedure could be improved upon, either through refinements or with a fundamentally different procedure. In particular, it may be possible to determine good heuristics for finding fast and efficient coder designs for a given application. A related research task is to better characterize the various aspects of complexity of coder designs. Such a characterization might allow a design procedure to take encoding and decoding speeds into account. In addition, variations in the underlying encoding and decoding procedures may yield speed improvements. Finally, we would like to identify trees (whether useful or not) that are well suited for inclusion in coder designs, and we would like to have a better understanding of useful trees. For example, we would like to identify new (or generalized) families of useful trees, and find a better characterization of useful trees.

ACKNOWLEDGMENTS

The work described was funded by the IPN-ISD Technology Program and performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

REFERENCES

1. A. Kiely, M. Klimesh, "A new entropy coding technique for data compression," *The InterPlanetary Network Progress Report 42-146, April–June 2001*, pp. 1–48, Jet Propulsion Laboratory, Pasadena, California, 2001. http://ipnpr.jpl.nasa.gov/progress report/42-146/146G.pdf
2. A. Kiely, M. Klimesh, "Memory-efficient recursive interleaved entropy coding," *The InterPlanetary Network Progress Report 42-146, April–June 2001*, pp. 1–14, Jet Propulsion Laboratory, Pasadena, California, 2001. http://ipnpr.jpl.nasa.gov/progress report/42-146/146J.pdf
3. P. G. Howard and J. S. Vitter, "Arithmetic coding for data compression," *Proc. IEEE* **82**, pp. 857–865, 1994.
4. S. W. Golomb, "Run-length encodings," *IEEE Trans. Inform. Theory* **IT-12**, pp. 399–401, 1966.
5. R. G. Gallager and D. C. Van Voorhis, "Optimal source codes for geometrically distributed integer alphabets," *IEEE Trans. Inform. Theory* **IT-21**, pp. 228–230, 1975.
6. G. G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Dev.* **28**, pp. 135–149, 1984.
7. J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM J. Res. Dev.* **23**, pp. 149–162, 1979.
8. J. Rissanen and G. G. Langdon, "Universal modeling and coding," *IEEE Trans. Inform. Theory* **IT-27**, pp. 12–23, 1981.
9. I. H. Witten, R. M. Neal, J. G. Cleary, "Arithmetic coding for data compression," *Comm. ACM* **30**, pp. 520–540, 1987.
10. W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Dev.* **32**, pp. 717–726, 1988.
11. A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inform. Sys.* **16**, pp. 256–294, 1998.
12. M. Boliek, J. D. Allen, E. L. Schwartz, M. J. Gormish, "Very high speed entropy coding," *Proc. IEEE Int. Conf. Image Proc. (ICIP-94)*, pp. 625–629, Austin, Texas, 1994.
13. L. Bottou, P. G. Howard, Y. Bengio, "The Z-coder adaptive binary coder," *Proc. IEEE Data Compression Conf.*, pp. 13–22, Snowbird, Utah, 1998.
14. P. G. Howard, "Interleaving entropy codes," *Proc. Compression and Complexity of Sequences 1997*, Salerno, Italy, pp. 45–55, 1998.
15. F. Ono, S. Kino, M. Yoshida, and T. Kimura, "Bi-level image coding with MELCODE — comparison of block type code and arithmetic type code," *Proc. IEEE Global Telecom. Conf. (GLOBECOM '89)*, pp. 0255 - 0260, Dallas, Texas, 1989.
16. K. Nguyen-Phi, H. Weinrichter, "A new binary source coder and its application to bi-level image compression," *Proc. IEEE Global Telecom. Conf. (GLOBECOM '96)*, pp. 1483–1487, London, England, 1996.